APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: **METHOD AND APPARATUS FOR VECTORED BLOCK-LEVEL CHECKSUM FOR FILE SYSTEM DATA INTEGRITY**

APPLICANT: **Jeffrey S. BONWICK and Matthew A. AHRENS**

32615
PATENT TRADEMARK OFFICE

"EXPRESS MAIL" Mailing Label Number: EV436026717US
Date of Deposit: April 21, 2004

METHOD AND APPARATUS FOR VECTORED BLOCK-LEVEL CHECKSUM FOR FILE SYSTEM DATA INTEGRITY

Background

[0001] A typical operating system includes a file system. The file system provides a mechanism for the storage and retrieval of files and a hierarchical directory structure for the naming of multiple files. More specifically, the file system stores information provided by the user (*i.e.*, data) and information describing the characteristics of the data (*i.e.*, metadata). The file system also provides extensive programming interfaces to enable the creation and deletion of files, reading and writing of files, performing seeks within a file, creating and deleting directories, managing directory contents, etc. In addition, the file system also provides management interfaces to create and delete file systems. File systems are typically controlled and restricted by operating system parameters. For example, most operating systems limit the maximum number of file names that can be handled within their file system. Some operating systems also limit the size of files that can be managed under a file system.

[0002] An application, which may reside on the local system (*i.e.*, computer) or may be located on a remote system, uses files as an abstraction to address data. Conventionally, this data is stored on a storage device, such as a disk.

[0003] To access a file, the operating system (via the file system) typically provides file manipulation interfaces to open, close, read, and write the data within each file. More specifically, the file system stores data on the storage device by managing the allocation of space within the storage device. Typically, the volume manager provides space which is managed by the file system. Two common types of file system space allocation strategies are known as block-based allocation and extent-based allocation. Block-based allocation creates incremental disk space for each file each time the file is extended (*i.e.*, modified via a write request to add information), whereas extent-based

allocation creates a large series of contiguous blocks (*i.e.*, extents) each time the file exhausts the space available in the file's last extent.

[0004] When allocating space, both block-based and extent-based allocation use space provided by the volume manager. The volume manager allows multiple physical disks to be used as a single volume (*i.e.*, a virtual disk) to provide larger consolidated storage sizes and simpler management. The volume manager allows users to organize data along volume boundaries (*i.e.*, each volume has physical disk space allocated to the volume such that the volume is tied only to that dedicated physical disk). The volume manager is typically implemented as a separate layer between the physical disks and the file system, and is presented to the user as a virtual disk device. In other words, volume managers organize the collections of physical devices (*e.g.*, disks) into virtual devices. Additionally, the space allocated within the volume manager is handled by the file system. Consequently, the volume manager is not aware of which blocks within the available storage space are in use and which blocks are free for data to be stored.

[0005] Further, file systems may be mounted on the virtual disk devices. Thus, physical disks are partitioned and allocated to multiple virtual disk devices, and each virtual disk device is capable of having a file system that exclusively uses that particular virtual disk device. A request to access a file is typically performed by an application, via the file system, using a file name and logical offset. This file name and logical offset (*i.e.*, the manner in which applications express file operation requests) corresponds to a location within the virtual disk device. Subsequently, the request is translated to physical disk space on the storage device by the volume manager, allowing the user of the application to access the data within a particular file.

[0006] Files systems and/or volume managers may include a mechanism for determining whether the data stored within the file system has been corrupted or otherwise altered. One such mechanism is a checksum. The checksum corresponds to a set of bits obtained by applying a particular formula (*e.g.*, Message Digest (MD) 5, Fletcher, Cyclic Redundancy Check (CRC) etc.) to the piece of data. The checksum for the

particular piece of data is then stored adjacent to the corresponding data in the file system (*e.g.*, using 520 byte sectors).

Summary

[0007] In general, in one aspect, the invention relates to a method for storing a data block, comprising storing the data block in a storage pool, obtaining a data block location, determining a checksum function for the data block, calculating a data block checksum using the checksum function for the data block, and storing a first indirect block in the storage pool, wherein the first indirect block comprises the data block location, the data block checksum, and a checksum function ID corresponding to the checksum function for the data block.

[0008] In general, in one aspect, the invention relates to a method for storing a first data block and a second data block, comprising storing the first data block and the second data block in a storage pool, obtaining a first data block location and a second data block location, calculating a first data block checksum for the first data block using a first checksum function, wherein the first checksum function is associated with a first checksum ID, calculating a second data block checksum for the second data block using a second checksum function, wherein the second checksum function is associated with a second checksum ID, and storing an array of block pointers in an indirect block, wherein the array of block pointers comprises a first block pointer comprising the first data block location, the first data block checksum, and the first checksum ID, and a second block pointer comprising the second data block location, the second data block checksum, and the second checksum ID.

[0009] In general, in one aspect, the invention relates to a method for retrieving data in a data block, comprising obtaining an indirect block comprising a stored checksum, a stored checksum function ID, and a data block location, obtaining the data block using the data block location, calculating the checksum for the data block using a checksum function corresponding to the stored checksum ID to obtain a calculated checksum, retrieving the data from the data block, if the stored checksum equals the calculated

checksum, and performing an appropriate action, if the stored checksum is not equal to the calculated checksum.

[0010] In general, in one aspect, the invention relates to a method for storing and retrieving a data block, comprising storing the data block in a storage pool, obtaining a data block location, determining a checksum function for the data block, calculating a data block checksum using the checksum function for the data block, storing an indirect block in the storage pool, wherein the indirect block comprises the data block location, the data block checksum, and a checksum function ID corresponding to the checksum function for the data block, obtaining the indirect block comprising the data block checksum, the checksum function ID, and the data block location, obtaining the data block using the data block location, calculating the checksum for the data block using a checksum function corresponding to the checksum ID to obtain a calculated checksum, retrieving the data from the data block, if the data block checksum equals the calculated checksum, and performing an appropriate action, if the data block checksum is not equal to the calculated checksum.

[0011] In general, in one aspect, the invention relates to a system for storing a data block, comprising a storage pool comprising the data block and the first indirect block, wherein the first indirect block comprises a data block checksum, a data block checksum function ID, and a data block location, and a storage pool allocator configured to store the data block and the first indirect block in the storage pool, and a plurality of checksum functions, wherein each of the plurality of checksum functions is associated with a checksum function ID.

[0012] In general, in one aspect, the invention relates to a computer system for storing a data block, comprising a processor, a memory, a storage device, and software instructions stored in the memory for enabling the computer system under control of the processor, to store the data block in a storage pool, obtain a data block location, determine a checksum function for the data block, calculate a data block checksum using the checksum function for the data block, and store a first indirect block in the storage pool, wherein the first indirect block comprises the data block location, the data

block checksum, and a checksum function ID corresponding to the checksum function for the data block.

[0013] In general, in one aspect, the invention relates to a network system having a plurality of nodes, comprising a storage pool comprising the data block and the first indirect block, wherein the first indirect block comprises a data block checksum, a data block checksum function ID, and a data block location, and a storage pool allocator configured to store the data block and the first indirect block in the storage pool, and a plurality of checksum functions, wherein each of the plurality of checksum functions is associated with a checksum function ID, wherein the storage pool is located on any one of the plurality of nodes, and wherein the storage pool allocator is located on any one of the plurality of nodes.

[0014] Other aspects of the invention will be apparent from the following description and the appended claims.

Brief Description of Drawings

[0015] Figure 1 shows a system architecture in accordance with an embodiment of the invention.

[0016] Figure 2 shows a storage pool allocator in accordance with an embodiment of the invention.

[0017] Figure 3 shows a hierarchical data configuration in accordance with an embodiment of the invention.

[0018] Figure 4 shows a flow chart in accordance with an embodiment of the invention.

[0019] Figure 5 shows a hierarchical data configuration in accordance with an embodiment of the invention.

[0020] Figure 6 shows a flow chart in accordance with an embodiment of the invention.

[0021] Figure 7 shows a computer system in accordance with an embodiment of the invention.

Detailed Description

[0022] Specific embodiments of the invention will now be described in detail with reference to the accompanying figures. Like elements in the various figures are denoted by like reference numerals for consistency.

[0023] In the following detailed description of one or more embodiments of the invention, numerous specific details are set forth in order to provide a more thorough understanding of the invention. However, it will be apparent to one of ordinary skill in the art that the invention may be practiced without these specific details. In other instances, well-known features have not been described in detail to avoid obscuring the invention.

[0024] In general, embodiments of the invention relate to a method and apparatus for using checksums in a file system. More specifically, embodiments of the invention provide a method and apparatus for implementing vectored block-level checksums in a file system. Further, embodiments of the invention provide a method and apparatus for storing the data and the corresponding checksum in different locations, thereby providing robust fault isolation. Further, embodiments of the invention provide a method and apparatus for selecting a checksum function on a per-block basis.

[0025] Figure 1 shows a system architecture in accordance with one embodiment of the invention. The system architecture includes an operating system (103) interacting with a file system (100), which in turn interfaces with a storage pool (108). In one embodiment of the invention, the file system (100) includes a system call interface (102), a data management unit (DMU) (104), and a storage pool allocator (SPA) (106).

[0026] The operating system (103) typically interfaces with the file system (100) via a system call interface (102). The operating system (103) provides operations (101) for users to access files within the file system (100). These operations (101) may include read, write, open, close, etc. In one embodiment of the invention, the file system (100) is an object-based file system (*i.e.,* both data and metadata are stored as objects). More specifically, the file system (100) includes functionality to store both data and

corresponding metadata in the storage pool (108). Thus, the aforementioned operations (101) provided by the operating system (103) correspond to operations on objects.

[0027] More specifically, in one embodiment of the invention, a request to perform a particular operation (101) (*i.e.,* a transaction) is forwarded from the operating system (103), via the system call interface (102), to the DMU (104). In one embodiment of the invention, the DMU (104) translates the request to perform an operation on an object directly to a request to perform a read or write operation at a physical location within the storage pool (108). More specifically, the DMU (104) represents the objects as data blocks and indirect blocks as described in Figure 3 below. Additionally, in one embodiment of the invention, the DMU (104) includes functionality to group related work (*i.e.,* modifications to data blocks and indirect blocks) into I/O requests allowing related blocks to be forwarded to the SPA (106) together. The SPA (106) receives transactions from the DMU (106) and subsequently writes the blocks into the storage pool (108). The operation of the SPA (106) is described in Figure 2 below.

[0028] In one embodiment of the invention, the storage pool (108) includes one or more physical disks (disks (110A-110N)). Further, in one embodiment of the invention, the storage capacity of the storage pool (108) may increase and decrease dynamically as physical disks are added and removed from the storage pool. In one embodiment of the invention, the storage space available in the storage pool (108) is managed by the SPA (106).

[0029] Figure 2 shows the SPA (106) in accordance with one embodiment of the invention. The SPA (106) may include an I/O management module (200), a compression module (201), an encryption module (202), a checksum module (203), and a metaslab allocator (204). Each of these modules is described in detail below.

[0030] As noted above, the SPA (106) receives transactions from the DMU (104). More specifically, the I/O management module (200), within the SPA (106), receives transactions from the DMU (104) and groups the transactions into transaction groups in accordance with one embodiment of the invention. The compression module (201) provides functionality to compress larger logical blocks (*i.e.,* data blocks and indirect

blocks) into smaller segments, where a segment is a region of physical disk space. For example, a logical block size of 8K bytes may be compressed to a size of 2K bytes for efficient storage. Further, in one embodiment of the invention, the encryption module (202) provides various data encryption algorithms. The data encryption algorithms may be used, for example, to prevent unauthorized access. In one embodiment of the invention, the checksum module (203) includes functionality to calculate a checksum for data (*i.e.,* data stored in a data block) and metadata (*i.e.,* data stored in an indirect block) within the storage pool. The checksum may be used, for example, to ensure data has not been corrupted.

[0031] Further, in one embodiment of the invention, the checksum module (203) may include a list of checksum functions that are available to the checksum module (203). Specifically, the checksum module (203) may use any of the available checksum components to calculate the checksum for a given block (data block or indirect block). In one embodiment of the invention, each checksum function is identified by a unique ID, *e.g.,* a 5-bit number, etc. The unique identifier may be stored with the corresponding checksum function in the checksum component, for example, in a table indexed by unique ID.

[0032] As discussed above, the SPA (106) provides an interface to the storage pool and manages allocation of storage space within the storage pool (108). More specifically, in one embodiment of the invention, the SPA (106) uses the metaslab allocator (204) to manage the allocation of storage space in the storage pool (108).

[0033] In one embodiment of the invention, the storage space in the storage pool is divided into contiguous regions of data, *i.e.*, metaslabs. The metaslabs may in turn be divided into segments (*i.e.,* portions of the metaslab). The segments may all be the same size, or alternatively, may be a range of sizes. The metaslab allocator (204) includes functionality to allocate large or small segments to store data blocks and indirect blocks. In one embodiment of the invention, allocation of the segments within the metaslabs is based on the size of the blocks within the I/O requests. That is, small segments are allocated for small blocks, while large segments are allocated for large

blocks. The allocation of segments based on the size of the blocks may allow for more efficient storage of data and metadata in the storage pool by reducing the amount of unused space within a given metaslab. Further, using large segments for large blocks may allow for more efficient access to data (and metadata) by reducing the number of DMU (104) translations and/or reducing the number of I/O operations. In one embodiment of the invention, the metaslab allocator may include a policy that specifies a method to allocate segments.

[0034] As noted above, the storage pool (108) is divided into metaslabs, which are further divided into segments. Each of the segments within the metaslab may then be used to store a data block (*i.e.*, data) or an indirect block (*i.e.*, metadata). Figure 3 shows the hierarchical data configuration (hereinafter referred to as a "tree") for storing data blocks and indirect blocks within the storage pool in accordance with one embodiment of the invention. In one embodiment of the invention, the tree includes a root block (300), one or more levels of indirect blocks (302, 304, 306), and one or more data blocks (308, 310, 312, 314). In one embodiment of the invention, the location of the root block (300) is in a particular location within the storage pool. The root block (300) typically points to subsequent indirect blocks (302, 304, 306). In one embodiment of the invention, indirect blocks (302, 304, 306) may be arrays of block pointers (*e.g.*, 302A, 302B, etc.) that, directly or indirectly, reference to data blocks (308, 310, 312, 314). The data blocks (308, 310, 312, 314) contain actual data of files stored in the storage pool. One skilled in the art will appreciate that several layers of indirect blocks may exist between the root block (300) and the data blocks (308, 310, 312, 314).

[0035] In contrast to the root block (300), indirect blocks and data blocks may be located anywhere in the storage pool (108 in Figure 1). In one embodiment of the invention, the root block (300) and each block pointer (*e.g.*, 302A, 302B, etc.) includes data as shown in the expanded block pointer (302B). One skilled in the art will appreciate that data blocks do not include this information; rather data blocks contain actual data of files within the file system.

[0036] In one embodiment of the invention, each block pointer includes a metaslab ID (318), an offset (320) within the metaslab, a birth value (322) of the block referenced by the block pointer, a checksum (324) of the data stored in the block (data block or indirect block) referenced by the block pointer, and a checksum function ID (326) corresponding to the checksum function used to calculate the checksum (324).

[0037] In one embodiment of the invention, each indirect block includes a metaslab ID (318), an offset (320) within the metaslab, a birth value (322) of the indirect block (*i.e.,* the transaction group in which the indirect block was committed), a checksum (324) of the data stored in the block (data block or indirect block) referenced by the indirect block,

[0038] In one embodiment of the invention, the metaslab ID (318) and offset (320) are used to determine the location of the block (data block or indirect block) in the storage pool. The metaslab ID (318) identifies a particular metaslab. More specifically, the metaslab ID (318) may identify the particular disk (within the storage pool) upon which the metaslab resides and where in the disk the metaslab begins. The offset (320) may then be used to reference a particular segment in the metaslab. In one embodiment of the invention, the data within the segment referenced by the particular metaslab ID (318) and offset (320) may correspond to either a data block or an indirect block. If the data corresponds to an indirect block, then the metaslab ID and offset within a block pointer in the indirect block are extracted and used to locate a subsequent data block or indirect block. The tree may be traversed in this manner to eventually retrieve a requested data block.

[0039] In one embodiment of the invention, copy-on-write transactions are performed for every data write request to a file. Specifically, all write requests cause new segments to be allocated for the modified data. Therefore, the retrieved data blocks and indirect blocks are never overwritten (until a modified version of the data block and indirect block is committed). More specifically, the DMU writes out all the modified data blocks in the tree to unused segments within the storage pool. Subsequently, the DMU writes out the corresponding block pointers (within indirect blocks) to unused segments

in the storage pool. In one embodiment of the invention, fields (*i.e.,* metaslab ID, offset, birth, checksum, checksum function ID) for the corresponding block pointers are populated by the DMU prior to sending an I/O request to the SPA. The indirect blocks containing the block pointers are typically written one level at a time. To complete the copy-on-write transaction, the SPA issues a single write that atomically changes the root block to reference the indirect blocks referencing the modified data block.

[0040] Using the aforementioned infrastructure shown in Figures 1-3, the following discussion describes a method for implementing vectored block-level checksums in a file system in accordance with one embodiment of the invention. Figure 4 shows a flow chart in accordance with one embodiment of the invention. Initially, the DMU receives a transaction from an application, the operating system (or a subsystem therein), etc. (ST 100). The DMU subsequently groups transactions into one or more I/O requests (ST 102). These I/O requests are subsequently forwarded to the SPA (ST 104).

[0041] In one embodiment of the invention, each I/O request includes one or more data blocks, and/or one or more indirect blocks. As noted above, the file system is stored on disk using a hierarchical structure including data blocks and indirect blocks. Thus, for a given set of transactions, the first I/O request includes the data blocks to be written to disk, while subsequent I/O requests include the corresponding indirect blocks containing one or more block pointers. Accordingly, the I/O request referenced in ST 104 includes data blocks.

[0042] Continuing with the discussion of Figure 4, the SPA, upon receiving the I/O request (including data blocks) from the DMU, writes the data blocks into the storage pool (ST 106). The SPA subsequently determines a checksum function to use when calculating the checksum for each data block written into the storage pool (ST 108). In one embodiment of the invention, the checksum function used for each data block is pre-determined. Alternatively, the checksum function used for each data block is determined dynamically.

[0043] The SPA subsequently calculates a checksum using the corresponding checksum function for each data block written into the storage pool (ST 110). In one embodiment,

the checksum module (203 in Figure 2) within the SPA is used to calculate the checksum for each data block written into the storage pool. The checksums are subsequently forwarded to the DMU (ST 112). The DMU then assembles the indirect blocks using the checksums (ST 114). Specifically, the DMU places the checksum for a given data block in the appropriate block pointer within the indirect block (*i.e.,* the parent indirect block of the data block). Next, the indirect blocks are forwarded to the SPA (ST 116). Those skilled in the art will appreciate that the aforementioned indirect blocks correspond to the indirect blocks that directly point (via the block pointers) to the data blocks (as opposed to indirect blocks that point to other indirect blocks).

[0044] Next, the SPA receives and subsequently writes the indirect blocks into the storage pool (ST 118). A determination is then made whether additional indirect blocks exist to write into the storage pool (*i.e.,* whether the last indirect block written to the storage pool corresponds to the root block) (ST 120). If no additional indirect blocks exist, then the method is complete. However, if additional indirect blocks exist, then the SPA calculates the checksum from each of the indirect blocks written into the storage pool (ST 122). The checksums for each of the indirect blocks is subsequently forwarded to the DMU (ST 124). Steps ST 114 through ST 126 are subsequently repeated until the root block is written into the storage pool.

[0045] Figure 5 shows a hierarchical data configuration in accordance with an embodiment of the invention. Specifically, Figure 5 shows a detailed view of the data blocks and indirect blocks resulting from using the method described in Figure 4, in accordance with one embodiment of the invention. The file system includes four data blocks (*i.e.,* 408, 410, 412, and 414). Each data block (408, 410, 412, 414) within the file system includes a corresponding checksum (CS408, CS410, CS412, CS414, respectively) and checksum function ID (CSFI408, CSFI410, CSFI412, CSFI414, respectively) stored in the corresponding block pointer (406A, 406B, 404A, 404B, respectively) within the parent indirect block (406, 404, respectively). Each of the block pointers (*i.e.,* 406A, 406B, 404A, 404B) is stored in an indirect block (*i.e.,* 404, 406). Each indirect block (404, 406) also has a corresponding checksum (CS404, CS406, respectively) and checksum function ID (CSFI404, CSFI406, respectively)

stored in corresponding block pointers in a parent indirect block. Specifically, block pointer (402A) includes the checksum (CS406) and checksum function ID (CSFI406) for the indirect block (406), and block pointer (402B) includes the checksum (CS404) and checksum function ID (CSFI404) for the indirect block (404). In this particular example, the indirect block (402) is referenced by a root block (400). The root block (400) includes the checksum (CS402) and checksum function ID (CSFI402) for the indirect block (402).

[0046] Figure 6 shows a flow chart in accordance with one embodiment of the invention. More specifically, Figure 6 details a method for reading data in accordance with one embodiment of the invention. Initially, a transaction to read data is received (ST 140). A checksum, corresponding checksum function ID and a location to the next block (*i.e.*, a data block or an indirect block) stored in the root block, is subsequently retrieved (ST 142). The location (*e.g.*, the metaslab ID and offset) is subsequently used to obtain the block (*i.e.*, the data block or the indirect block) (ST 144). The checksum of the retrieved block is then calculated using the checksum function corresponding to the checksum function ID (ST 146). A determination is subsequently made whether the stored checksum is equal to the calculated checksum (ST 148). If the stored checksum is not equal to the calculated checksum, then an appropriate action is performed (*e.g.*, an error message is generated indicating that the data is corrupted) (ST 156).

[0047] If the stored checksum is equal to the calculated checksum, then a determination is made whether the retrieved block is the data block (ST 150). If the retrieved block is the data block, then the data is extracted from the data block and presented to the process requesting the data (ST 152). Alternatively, if the retrieved block is not the data block, then the location, checksum, and checksum function ID for the next block (stored within the retrieved block) are obtained (ST 154). The block (data block or indirect block) at the location is subsequently obtained (ST 158). Steps 144 through 154 are subsequently repeated until either data corruption is encountered (*i.e.*, ST 148) or the data block is encountered (*i.e.*, ST 152).

[0048] The invention may be implemented on virtually any type of computer regardless of the platform being used. For example, as shown in Figure 7, a networked computer system (500) includes a processor (502), associated memory (504), a storage device (506), and numerous other elements and functionalities typical of today's computers (not shown). The networked computer (500) may also include input means, such as a keyboard (508) and a mouse (510), and output means, such as a monitor (512). The networked computer system (500) is connected to a local area network (LAN) or a wide area network (*e.g.,* the Internet) (not shown) via a network interface connection (not shown). Those skilled in the art will appreciate that these input and output means may take other forms. Further, those skilled in the art will appreciate that one or more elements of the aforementioned computer (500) may be located at a remote location and connected to the other elements over a network. Further, the invention may be implemented on a distributed system having a plurality of nodes, where each portion of the invention (*e.g.,* the storage pool, the SPA, the DMU, etc.) may be located on a different node within the distributed system. In one embodiment of the invention, the node corresponds to a computer system. Alternatively, the node may correspond to a processor with associated physical memory.

[0049] While the invention has been described with respect to a limited number of embodiments, those skilled in the art, having benefit of this disclosure, will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.